# A PASCAL PACKAGE FOR CREATION, MAINTENANCE AND SEARCH IN A SMALL BIBLIOGRAPHIC INFORMATION RETRIEVAL SYSTEM

COSC460 RESEARCH PROJECT

MR. PATRICK LAU CHEN CHAI

UNIVERSITY OF CANTERBURY

5TH OCTOBER, 1981

SUPERVISOR: DR. WOLFGANG KREUTZER

ACKNOWLEDGEMENT

I would like to extend my appreciation to Dr. Wolfgang Kreutzer for his invaluable assistance to me in carrying out the project especially in the area of recommending references to read and checking the progress of the project from time to time.

I would also like to thank Miss Sarah Frampton for assisting me in the typing of this report.

Patrick Lau Chen Chai
5th October, 1981.

CONTENTS

# 1. INTRODUCTION

This project focusses on the design, implementation and test of a small experimental information retrieval system for bibliographical data. The primary aim of the project is to experiment with a few different data structures which can be used to design the system and eventually deciding on a final design which allows fast access to documents and efficient usage of storage space.

This report will describe the different data structures used for storing the thesaurus, descriptor-records and references. The results obtained from the analysis of the performance of each design will be shown in the report. A brief discussion on why the final design was chosen will also be carried out. The last section of the report will concentrate on the actual design implemented, including a brief documentation on how the package works and the various facilities provided.

## 2. DATA STRUCTURES

There are a variety of ways to design an information retrieval system but one of the most common techniques used is to maintain a dictionary of all keywords of documents. This is mainly due to the reason that it is easier and faster to search through the dictionary to see if a particular keyword is present than to search through each document for the keyword. The dictionary which this system uses is called a thesaurus where synonyms of keywords of documents are also included in the dictionary.

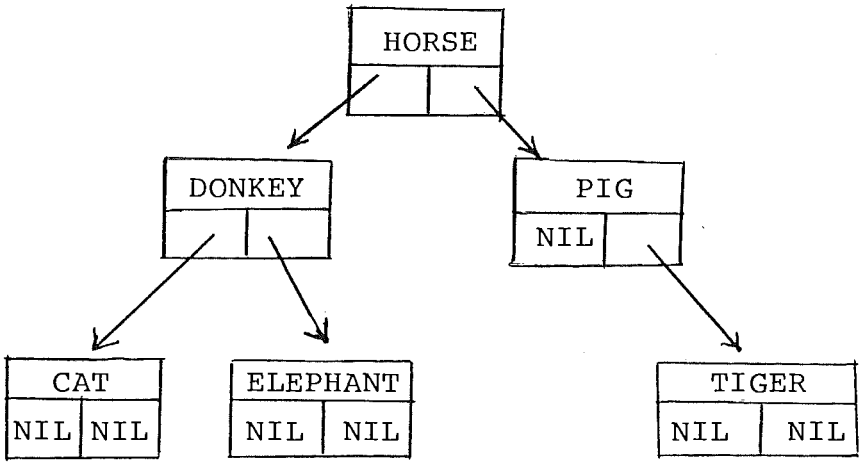
So, the first component to design is the thesaurus. Three methods of organising the descriptors were examined, namely a) a sorted linear list, b) a sorted tree and c) a hash table. A detailed description of each method is given below.

### a) Sorted linear list

This is perhaps one of the easiest ways of maintaining the thesaurus. A binary search on the list of descriptors can be rapidly carried out since the list is always sorted. Before any descriptor can be added to the thesaurus, a search has to be done on the current list to see whether the descriptor is already present or not. No duplicate descriptors are allowed in the thesaurus. All new descriptors of the current document being inserted are added to the bottom of the current list and then a sort is performed on these new descriptors with the rest of the descriptors in the thesaurus. The sorting algorithm used is the bubble-sort because using this method, we can avoid performing another sort on the current list which is already sorted.

b) Lexicographic tree

Another method of storing descriptors in the thesaurus is by creating a lexicographic tree of descriptors




A LEXICOGRAPHIC TREE

DIAGRAM 1

Whenever we want to add a new descriptor, we have to traverse down the tree from the root and check whether the descriptor is already present. If not, the new descriptor is attached to the tree at the appropriate position so that the lexicographic nature of the tree is maintained.

c) Hash tables

The third and last method tested for storing descriptors is by hashing the descriptors to obtain an address to a table, containing other descriptors. There is always the possibility of collisions whenever hashing method is used and so an auxiliary table is maintained to store colliding descriptors. The method used to locate the next available space in the auxiliary table to store a colliding descriptor is by performing a simple linear search down the table till we encounter an empty space. Appropriate links are maintained to allow searches to be performed on the descriptors.

The main purpose of any information retrieval system is to be able to retrieve relevant information at a fast speed. Relevant documents must be able to be located quickly and easily. The most common strategy used is to maintain a list of addresses of documents of each descriptor in either a matrix or an inverted file which I will call the document-descriptor file component of the system. This is a more efficient method to retrieve all the relevant documents for a particular descriptor than to scan through the whole document data base to locate the relevant documents. Less searches would be required and this is particularly evident when compound queries using simple boolean operators like AND, OR and NOT are permitted.

The two possible data structures of the document-descriptors are as follows:

a) Boolean matrix

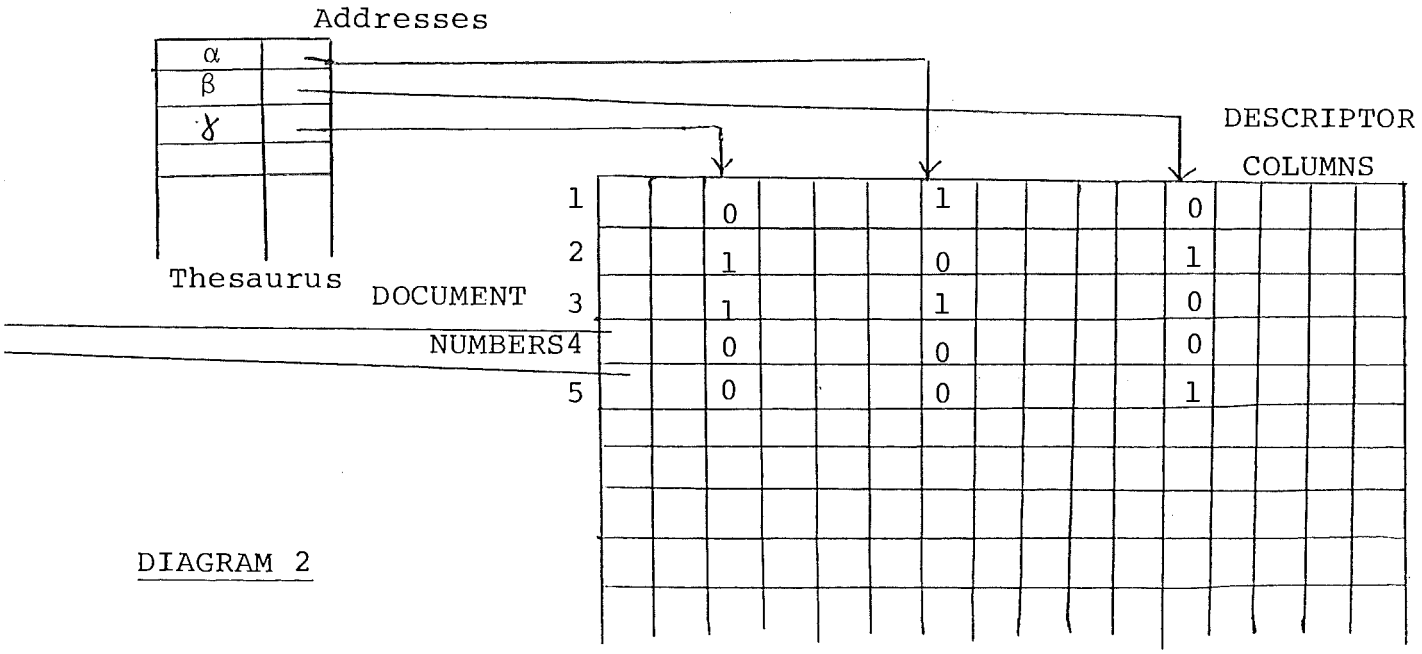
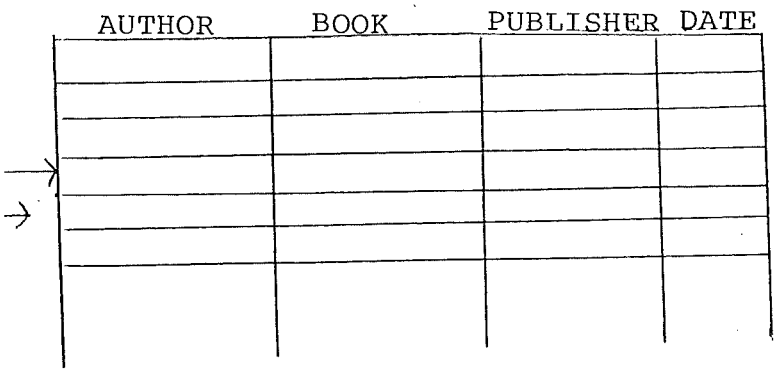



DIAGRAM 2

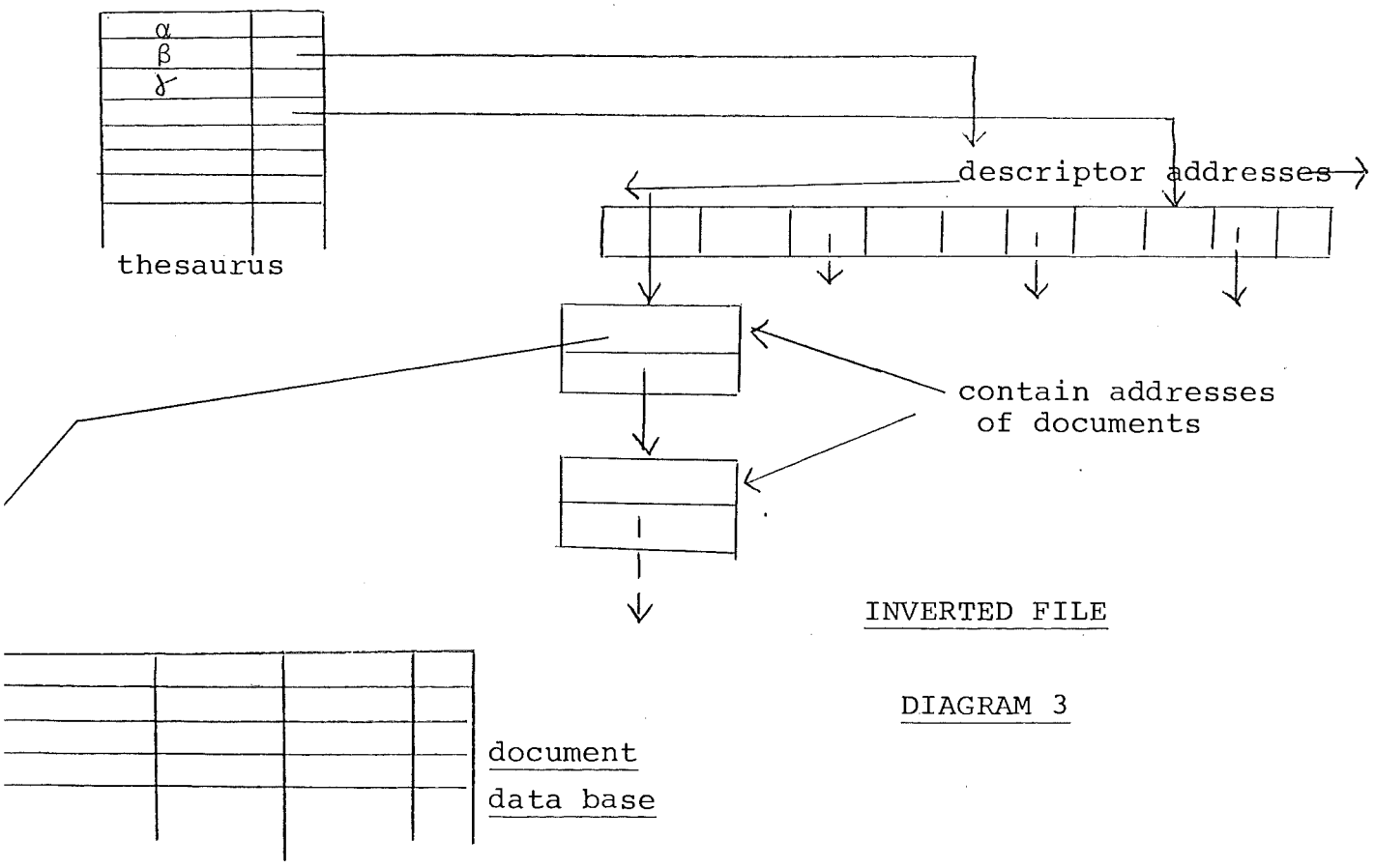
BOOLEAN MATRIX




DOCUMENT DATA BASE

Looking at diagram 2, you can see that it is quite a trivial matter to retrieve relevant documents for any particular query. Associated with each descriptor in the thesaurus is the column number for the descriptor in the boolean matrix. Each row of the boolean matrix denotes a document. You will notice that no descriptor or document details need be stored in the boolean matrix because the two numbers are used as addresses of actual documents in the document data base while each column represents a descriptor store in the thesaurus. A 'true' condition is placed in each element of the matrix if the descriptor is part of the document. Otherwise, it is set to 'false'. So, scanning down a particular column and picking out elements with the 'true' condition will allow us to retrieve all relevant documents associated with this particular descriptor. It will also become intuitively obvious why it is easy to answer compound queries using boolean operators like AND, OR and NOT, using the boolean matrix.

b) Inverted file




INVERTED FILE

DIAGRAM 3

The second alternative for creating a document-descriptor file is by creating an inverted file. The design implemented uses linked lists to create the inverted file. Each node contains an address of the document in the document data base and a link to subsequent nodes which have the same descriptor in their documents. A one-dimensional pointer array is used to denote the starting location of each descriptor in the inverted file. It is also a rather simple design to implement.

Before I proceed to analyse the performances of each of the designs mentioned, let me say briefly about the way the document data base is constructed.

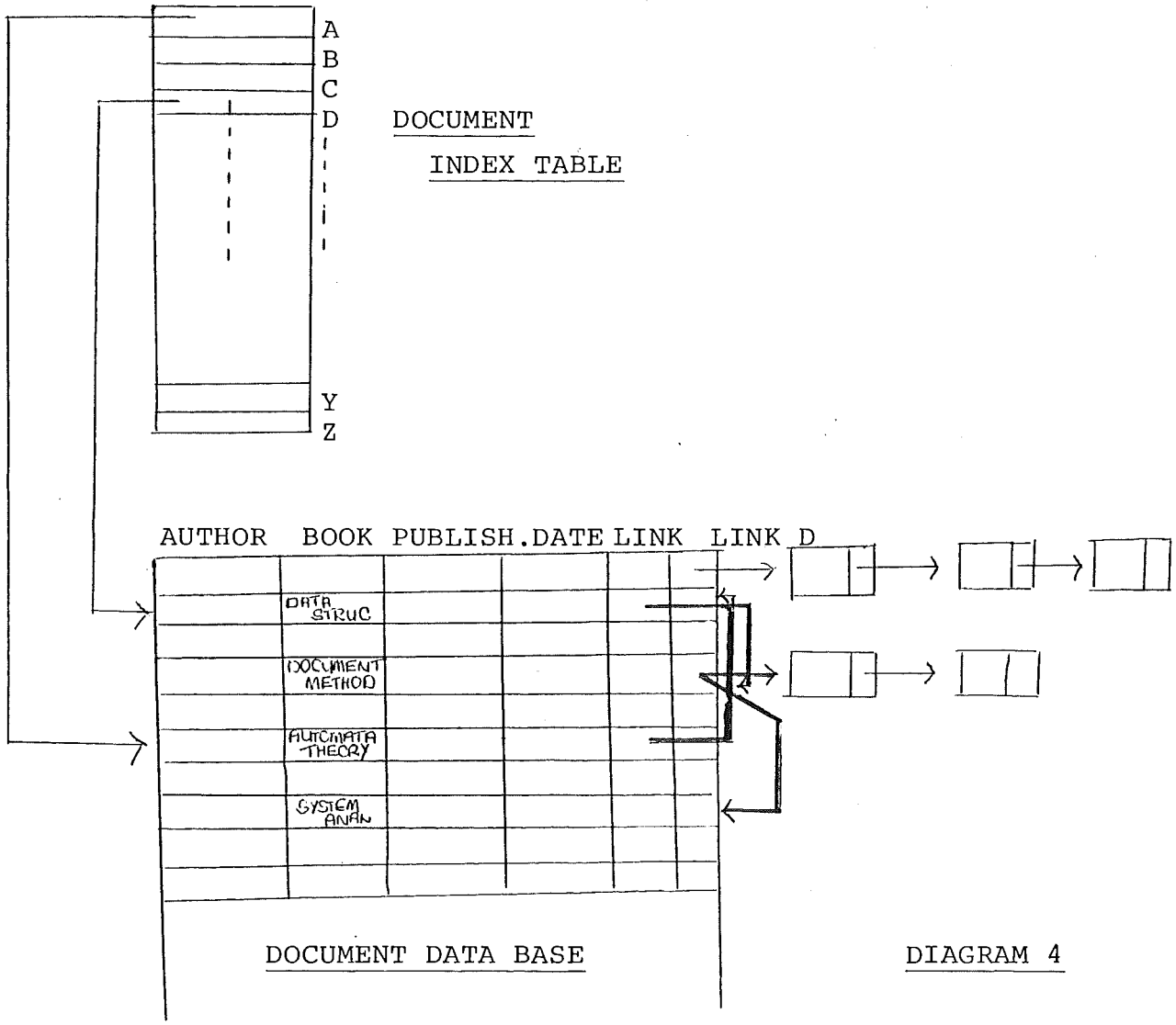



DIAGRAM 4

The document data base is unsorted since new documents are placed into the next available row as they come. Since descriptors are used to identify documents, the documents are maintained in alphabetical order by document names through pointers i.e. link field. The main reason for not sorting the documents is because details of documents are usually large and so it is rather ineffficient to sort them each time a new document is inserted into the system.

In order to facilitate the maintenance of the documents in sorted order through pointers, an index table is created. Each document name starts with a letter from A to Z and this property is used to store addresses of first document of each letter, into the index table. Hence, we do not have to go through the linked list pointers from the beginning to locate the appropriate position to insert pointers to the new document. Search time can be reduced substantially since only the relevant section of the data base needs to be searched.

Associated with each document is a linked list of descriptors of the document. This list is essential when it comes to deletion of documents because we need to delete descriptors at times. Besides, it provides a useful facility for users who want to know what are the descriptors of a particular document, especially so when a thesaurus is used i.e. to locate synonyms.

## 3. Analysis of Statistical results

Using a small sample of 125 documents, the following results were obtained as tabulated in APPENDIX I and II. Clearly, one can quickly pick the best method for designing the thesaurus. Let me attempt to explain why we obtained such performances.

### a) Insertion of new descriptors

To insert a new descriptor into the thesaurus, one has to perform a search to check if the descriptor is already in the thesaurus. For a sorted list, the search can be performed using binary search which achieves a search time that is O(logN).

But one has to sort the list again whenever new descriptors are added to the thesaurus. This is why so many comparisons are taken to insert a new descriptor into a sorted list.

On the other hand, the comparisons taken to insert a new descriptor into a lexicographical tree is very much smaller in comparison with a sorted list. This is attributed mainly to the reason that one does not have to sort the tree again whenever a new descriptor is added to the tree. All one has to do is traverse down the tree to check whether the descriptor is present and if not, attach this new descriptor to the tree at the appropriate position. Search time for a lexicographical tree depends on the depth of the tree i.e. $0(\log_2 N)$ for a balanced binary tree or the worst possible case 0(N) when the tree is actually a linear list.

Hashing descriptors into table locations proved to be the best method for inserting descriptors into the thesaurus. One does not have to worry about sorting the descriptors in alphabetical order. A search for a free storage location on the main hash table is first performed and this takes only one comparison. The descriptor is stored in the auxiliary table if descriptor is not found in main or auxiliary table and the hashed location in the main table is already occupied. From the results obtained, it is clear that more comparisons are needed as more descriptors are present in the thesaurus since the chances of more collisions increase with the number of descriptors in thesaurus. Nevertheless, it is still better than the other two methods.

b) Query and deletion of descriptors

For both query and deletion operations, a search on the thesaurus is required. Already I have mentioned that searching a sorted list takes 0(logn) and a sorted tree, on the depth of the tree but does not depend on n for the hash table. The number of comparisons required to locate a term within a hash table depends only on the load factor i.e. M/N where N is the total possible entries in the main hash table and M is the current number of descriptors in the thesaurus. The smaller the ratio, the less the possibility of collisions.

When deleting a descriptor from a sorted list or a sorted tree, the list and tree have to maintain their sorted order. This certainly takes up some overheads because for a sorted list, entries have to be moved to delete the descriptor and maintain the list in sorted order while for a sorted tree, the subtree rooted at the node being deleted must be reattached to the tree. Deleting a descriptor from a hashed table is easy. We just remove the descriptor from the table and if its collision bit is 1, maintain it so that when a search for a term with the same hash value is made, it will be clear that it needs to search the auxiliary table for the descriptor.

## 4. Criteria for design selection

For any information retrieval systems, the main objective is to allow fast retrieval of documents. This would mean that we need a fast searching method to look for our descriptors in the thesaurus. The obvious choice for the thesaurus would be to use the hashing strategy. Not only does this method provide fast access to descriptors, it also enables us to delete and insert new descriptors into the thesaurus without much overheads. The main disadvantage of using hashing is that we cannot get a sorted list of all the descriptors easily since they are not maintained in any sorted manner. Users can get a dump of all descriptors in the thesaurus in a sorted order. All the descriptors are placed in a table and a sort is performed to give the sorted list.

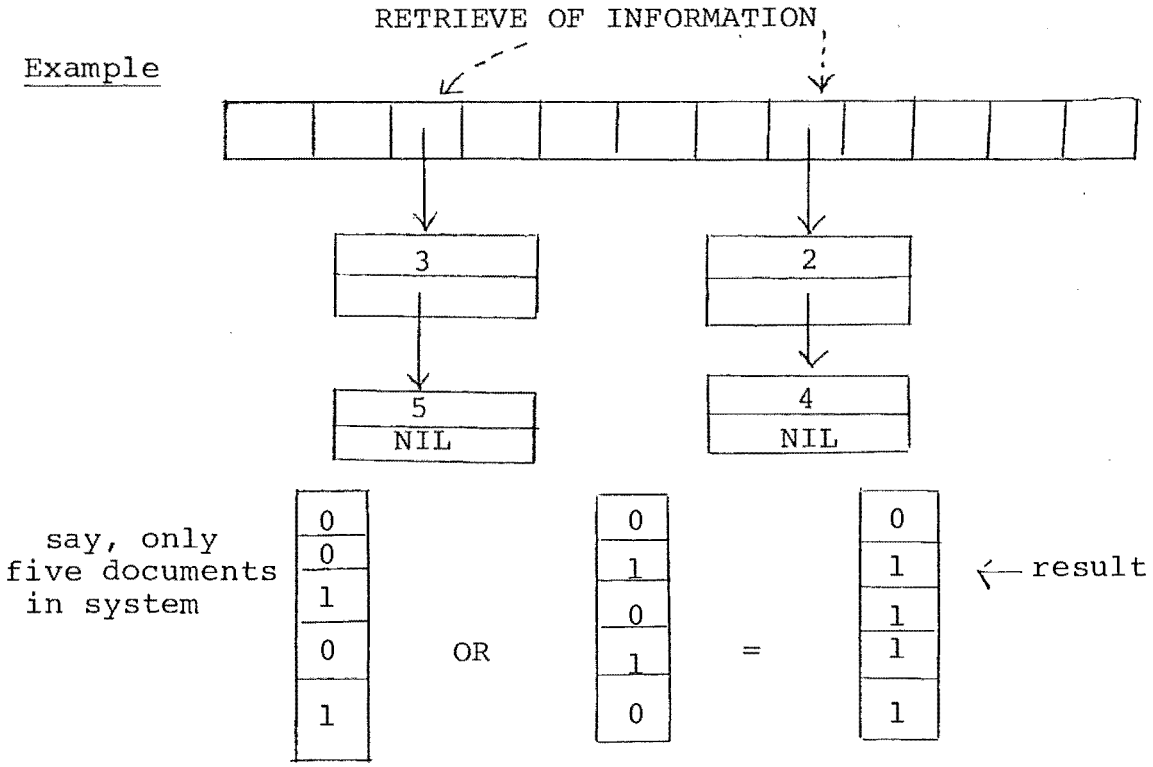
Memory space is always expensive and one has to find efficient ways of storage. Since we must cater for boolean-type queries e.g. RETRIEVE AND INFORMATION, it would seem that using the boolean matrix to store the document-description file is a convenient and easy way to answer such queries. On the other hand, the matrix uses up too much storage space especially when it is storing redundant information. All we want to know is which documents have a particular descriptor. All the elements of the matrix with a 'false' value stored in it can be eliminated since they are of no value to us.

For the inverted file, we have a linked list of document numbers for each descriptor. So, only relevant information is stored. No doubt we need extra space to store the pointers but this extra overhead is negligible when we realise the enormous savings we get by not storing all the elements with 'false' values.

Since I realised the advantage of having a boolean matrix to cater for boolean-type queries, what was implemented is a dynamic matrix, created specifically to entertain queries. All descriptors must be in the inverted file for valid query commands. The desired descriptor is first located and it contains all the document numbers for that descriptor. A new column is assigned to each descriptor and the document numbers are used as index of the column. Boolean operations are then performed to all the columns created to get the result.

This method entails some overheads because we need to transform descriptors in the inverted file into the boolean matrix. The tradeoff is that a boolean matrix permits fast search especially when we must allow boolean operators in query commands. This method is very advantageous for large data base systems which have a complex inverted file. Searching through the inverted file to execute the boolean operations would be rather inefficient.

Below is a diagram to describe how the columns are created.




So, four documents satisfy the query

Using a thesaurus to assist in the retrieval of information is certainly a good idea. But the success of the thesaurus depends mainly on the indexer/user knowing what synonyms to give for the descriptors. The other difficulty is that the system would treat say SYSTEM and SYSTEMS as different descriptors. So, I decided that the thesaurus should only contain root words of descriptors/synonyms.

Example

RETRIEVE is the root word for

1) RETRIEVES
2) RETRIEVING
3) RETRIEVAL
4) RETRIEVED.

So, again it is the job of the indexer to pick the appropriate root word to use as a descriptor of a document. This implies that users have to use root words of descriptors for their query instructions.

e.g. RETRIEVE .AND. SYSTEM
instead of
RETRIEVING .AND. SYSTEMS etc.

This may not sound like a good idea but it certainly cuts down the size of the thesaurus. The main advantage of using such a rule is that all documents containing the root word of the descriptor will be located.

## 5. Facilities provided by package

So far, I have only been talking about the different data structures available to store descriptors in the thesaurus and the document-descriptor file. Then, I went on to explain why I chose the final design for this system. Now, I would like to briefly say what are the facilities provided by the system and how one can use it.

This system is a batch system and so users have to make sure that they get their instruction commands correctly for each run. It is not difficult to convert this into an interactive system at all. A thesaurus is provided for the system but it is the task of the indexer to create synonyms of descriptors.

Inserting new documents into the data base can be rather tedious. Whenever you insert a new document, you have to insert all the descriptors and their synonyms to the system too. But, it is much easier to delete documents. All one has to do is to give the title of the document and it will be deleted from the system. On top of that, any descriptors referencing this document only, will be deleted from the thesaurus automatically by the system.

As mentioned earlier, users can perform one level boolean-type queries to retrieve the relevant documents. Simple one-level parenthesis can be used to assist query formulation e.g. (INFORMATION .AND. RETRIEVE) .OR. (.NOT. SYSTEM .OR. DATA). Each query is restricted to one simple input line only. Like most systems, the total number of documents found is returned to the user for each query and it is then up to the user whether he wants to have a listing of all the relevant documents which can be done by order of titles or authors.

The thesaurus, index table and document data base can be dumped to the printer if required. The listing of the document data base dump can be in alphabetical order of titles of documents or just as they are in the data base.

Another facility provided is error-checking. The relevant error messages will be printed whenever an error is encountered. Whenever a new document is inserted into the data base, the system checks that the length of each input detail is correct. Otherwise, it would flag out an error and ignore the rest of the input for this document. Error messages would also be flagged out if descriptors specify in query commands are not present in the system, invalid boolean operators used, unmatching parenthesis and a few others as outlined in the appendix. Refer to appendix for more documentation on how the package works.

## 6. Conclusions

Because the system is for a small bibliographic information system, it is quite difficult to get any realistic statistics about the effectiveness of a system using a particular data structure. What this project achieved was to present a few ways of designing an information retrieval systems and outlining their weaknesses as well as their strengths. Naturally, quite a substantial amount of work is still needed if we want to implement this system in real-time and using a larger data base file. The aim of the project was to write a package for a small bibliographic information retrieval system and this has been achieved using the most efficient design in terms of time and space as experimented in the project.

END

# APPENDICES

# APPENDIX I

## 1) DELETION

### AVERAGE COMPARISONS TAKEN

| #DOCUMENTS | #DESCRIPTORS | LINEAR LIST | TREE | HASH |
|---|---|---|---|---|
| 7 | 23 | 4 | 5 | 1 |
| 25 | 72 | 6 | 7 | 1 |
| 50 | 114 | 7 | 11 | 1 |
| 75 | 149 | 7 | 12 | 2 |
| 100 | 189 | 8 | 12 | 2 |
| 125 | 224 | 8 | 13 | 2 |

## 2) INSERTION

### AVERAGE COMPARISONS TAKEN

| #DOCUMENTS | #DESCRIPTORS | LINEAR LIST | TREE | HASH |
|---|---|---|---|---|
| 7 | 23 | 10 | 5 | 1 |
| 25 | 72 | 20 | 8 | 2 |
| 50 | 114 | 22 | 9 | 2 |
| 75 | 149 | 25 | 9 | 3 |
| 100 | 189 | 30 | 10 | 5 |
| 125 | 224 | 32 | 10 | 6 |

3) QUERY

AVERAGE COMPARISONS TAKEN

| #DOCUMENTS | #DESCRIPTORS | LINEAR LIST | TREE | HASH |
|---|---|---|---|---|
| 7 | 23 | 4 | 4 | 1 |
| 25 | 72 | 7 | 7 | 1 |
| 50 | 114 | 7 | 10 | 1 |
| 75 | 149 | 7 | 10 | 1 |
| 100 | 189 | 8 | 11 | 1 |
| 125 | 224 | 8 | 11 | 1 |

i) For any required operations, the user must first specify the necessary operation code with the following format

| OPCODE OPNUM |
|---|

where OPCODE can be

D - Delete a document
I - Insert a new document
P - Listing of tables/results
Q - Query command

and OPNUM must be given a value of zero for all operations except for OPCODE = P.
OPNUM is an integer number to indicate what type of listing is required. The user can only use the following numbers for OPNUM

1 - Dump document data base
2 - Dump document data base sorted by titles
3 - Dump document index table
4 - Print statistics gathered for insertion
5 - Print time taken to execute instruction
6 - Dump thesaurus in alphabetical order
7 - Dump hash and auxiliary tables
* 8 - 10 System operations to produce listings of relevant documents sorted either by authors or titles.

* These range of numbers are only accessible by the system. The system will set the relevant code for listing when user specify whether they want the listing sorted by author or title.

NOTE A blank is needed between OPCODE and OPNUM.

ii) Insertion of new documents

There are four fields which the user has to supply whenever he wants to insert a new document. They have to be input on separate lines and their lengths must not exceed the width limits set for each field. The system will reject all the input for a new document whenever an error occurs i.e. input exceed width limit. An error message will inform users which field is incorrectly inputed.

The following fields have to be specified in the order shown below, starting in column 1.

| | | | | |
|---|---|---|---|---|
| a) | Author | WIDTH | 30 | CHARACTERS |
| b) | Title of book | " | 55 | " |
| c) | Publisher | " | 40 | " |
| d) | Date Published | " | 10 | " |

After supplying these details of each document, the list of descriptors/synonyms has to be input next. Only one line of descriptors are permitted and each descriptor must be separated by at least a blank. The length of descriptors must not exceed 20 characters and a delimiter '*' must be tagged to the last descriptor (separated by at least a blank) to indicate termination of descriptors.

Below is an example of an insertion.

```
I O
PATRICK LAU
INFORMATION RETRIEVING METHODS
CANTERBURY PUBLISHERS
1981
INFORM RETRIEVE METHOD TECHNIQUE *
```

iii) Deletion of document

All the user needs to specify is the name of the document which must not be longer than fifty-five characters. The system will automatically delete any descriptors which reference this document only. That means, user is not permitted to delete descriptors from the system without first deleting the documents. Below are some examples of query commands

Example

```
D O
INFORMATION RETRIEVAL SYSTEMS
<<< DOCUMENT DELETED FROM SYSTEM >>>

D O
DATA BASE MANAGEMENT SYSTEMS

<<< DOCUMENT DELETED FROM SYSTEM >>>
```

iv) Query commands

Users can formulate queries to some degree of complexity using boolean operators and parenthesis. Only three boolean operators are allowed to be used and each must be enclosed by the delimiter '.'

e.g. .AND. .OR. .NOT.

The way the query command is executed follows the normal boolean algebra rules. No nesting of parenthesis are permitted. Only one level parenthesis are allowed. The delimiter '*' has to be tagged to the last descriptor, separated by at least a blank, to indicate the end of query command.

For each query command, the system will return the total number of documents located. There are two ways to list all the relevant documents found. The user has to input the required code to get the listing. The codes are as shown below.

(i) 'A' - list all documents sorted by author's names.

(ii) 'B' - list all documents sorted by document titles.

Examples

(i) Q O

RETRIEVE .AND. SYSTEM *

A

- all relevant documents found are to be listed in alphabetical order of author's names.

(ii) Q O

.NOT. INFORM .OR. (SEARCH .OR. SORT) *

B

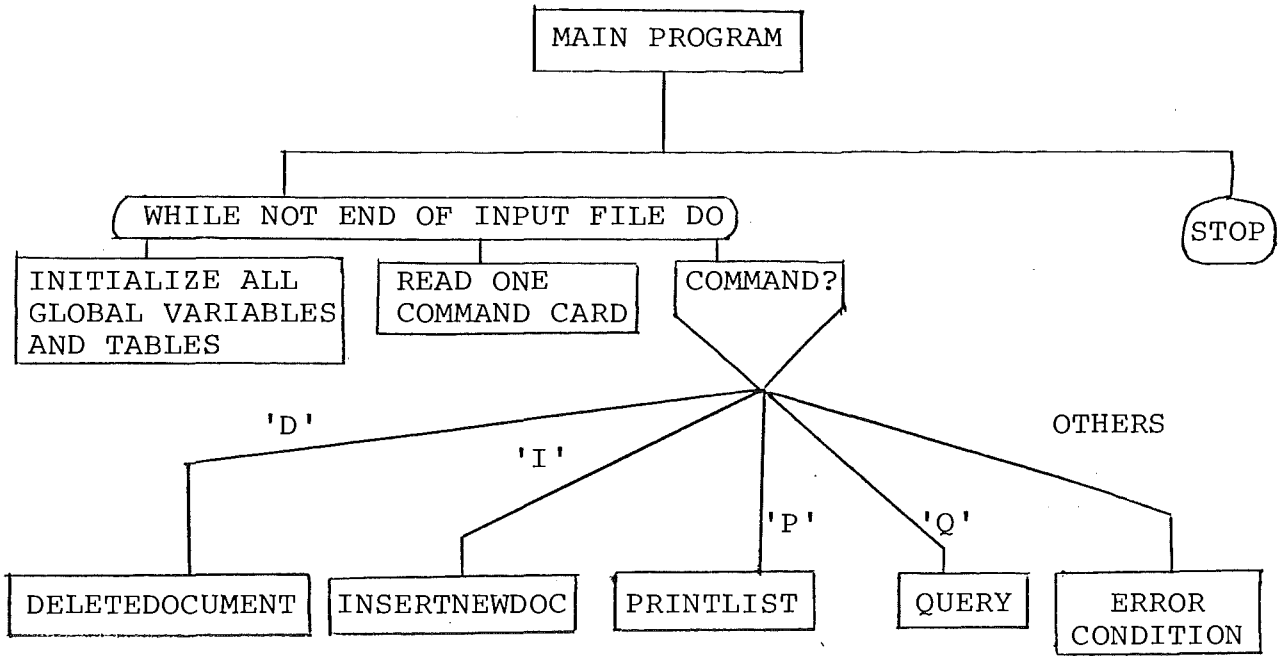
- all relevant documents found are to be listed in alphabetical order of document titles.

Note

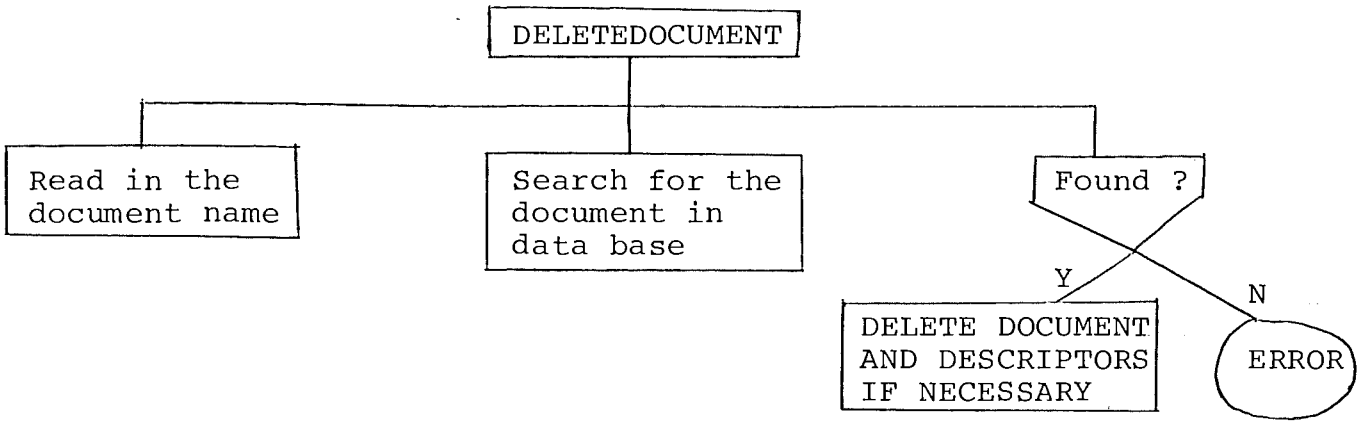
All input must start in column 1.

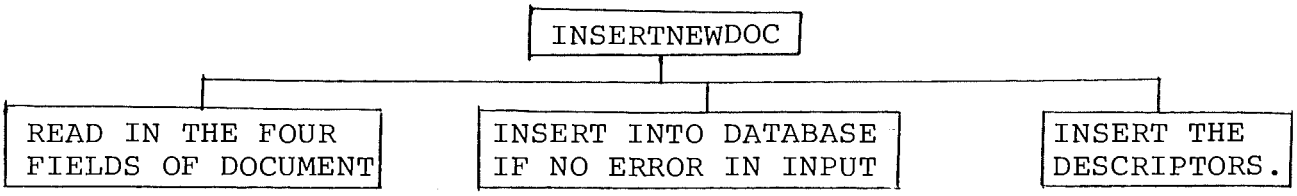
## A) PROGRAM STRUCTURE

Below is a structured diagram of the main program




For each command, the required procedure to perform the operation is called. Control returns to the main program after each execution and program terminates when it has finished reading all the input cards.

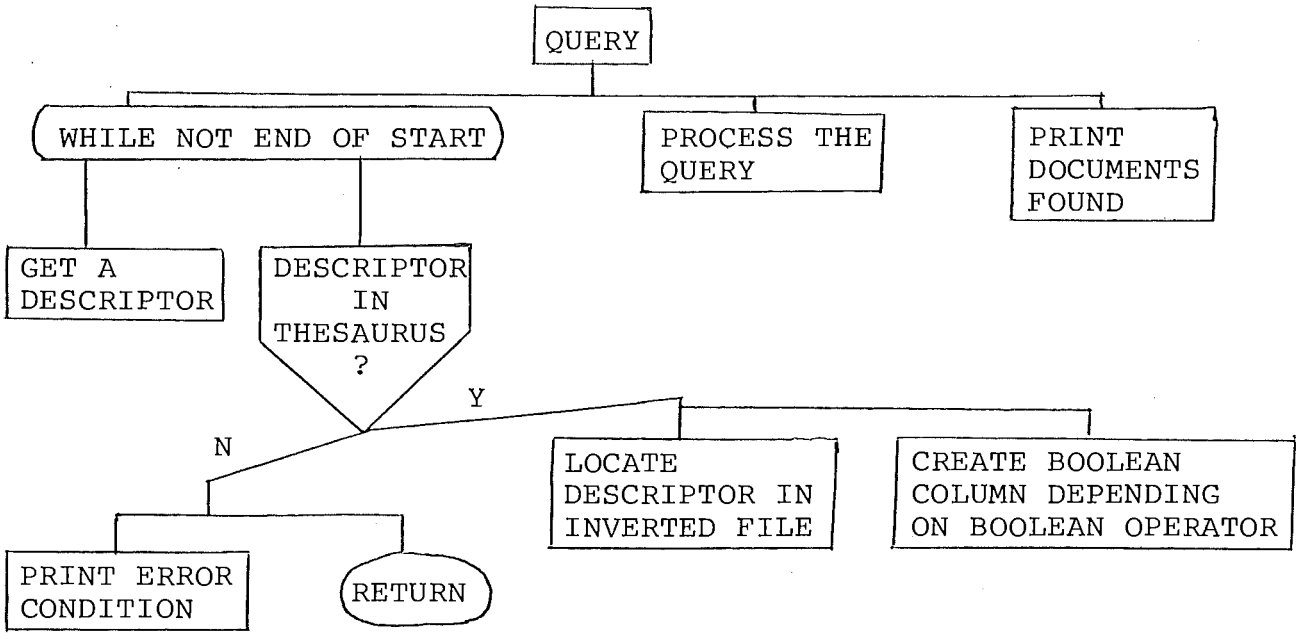





The procedure READDOCUMENT is called to read in the details of each document. For each field read in, it calls procedure EDITLINE to check that the field is correctly input i.e. does not exceed width limit.

Whenever a new document is inserted into the data base, it calls procedure INSERTDOCUMENTS to update the index table if necessary and retrieve the starting position to perform the search for the position to insert the document i.e. changing the pointers. Procedure SORTDOCUMENT performs the search along the links to attach links to the new document.

The program structure of the procedure PRINTLIST is basically that of a CASE statement. It branches to the relevant label and performs the operation specified under the label. This procedure caters for dumping and listings of documents, descriptors and statistics.



Among all the operations, the program structure for query is the most complex. Procedure QUERY mainly initializes the variables to be used and later on print all documents found by calling procedure PRINTLIST. SCANNER is the main procedure for query, which is called by procedure QUERY.

The main task of SCANNER is to transform a descriptor with the relevant numbers in the inverted file, into a boolean column. It then performs the boolean operation to the column if it is a .NOT. or .AND. operation. ALL .OR. operations, if any, will be done when all the columns have been created. This is to satisfy the precedence rule of operators.

The required procedures are called by SCANNER to perform he necessary task. For instance, procedure GETDESCRIPTOR gets a descriptor from the query command, performs a search on the thesaurus to check whether it is there and then retrieves the descriptor from the inverted file. So, the control structure is to pass control from procedure to procedure.

Whenever an error occurs in the input, it calls ERROREXIT which will flag the error condition and prints the error message. The rest of the input command is ignored.

B) DATA STRUCTURE

The data structure of the thesaurus consists of arrays to store the main hash table, auxiliary table, the addresses of the descriptors in the inverted file and the links for descriptors with some hashed values. The document data base is basically a table where each column represents a document record. Pascal allows RECORD declaration and each record consists of arrays for each field except the link fields. The descriptors of each document are stored as linked list to each record. Linked list structure is also used to represent the inverted file. Each descriptor has a linked list of document numbers relevant to the descriptor.

NOTE

All program documentation is done in the program itself.

REFERENCES

Information Retrieval: Computational and Theoretical aspects. Heaps, H.S. Academic Press Inc., 1978.

Data types and structures, Gotlieb, C.C. and Leo R. Gotlieb. Prentice-Hall Inc. Englewood Cliffs, N.J., 1978.

Algorithms and Data Structures = Programs. Niklaus Wirth. Prentice-Hall Inc. Englewood Cliffs. N.J., 1976.